Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results

Wolf-Dietrich Weber and Anoop Gupta
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

[redacted]

November 16, 1988

Abstract

A fundamental problem that any scalable multiprocessor must address is the ability to *tolerate* high latency memory operations. This paper explores the extent to which multiple hardware contexts per processor can help to mitigate the negative effects of high latency. In particular, we evaluate the performance of a directory-based cache coherent multiprocessor using memory reference traces obtained from three parallel applications. We explore the case where there are a small fixed number (2-4) of hardware contexts per processor and the context switch overhead is low. In contrast to previously proposed approaches, we also use a very simple context-switch criterion, namely a cache miss or a write-hit to shared data. Our results show that the effectiveness of multiple contexts depends on the nature of the applications, the context switch overhead, and the inherent latency of the machine architecture. Given reasonably low overhead hardware context switches, we show that two or four contexts can achieve substantial performance gains over a single context. For one application, the processor utilization increased by about 65% with two contexts and by about 100% with four contexts.

Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results

Wolf-Dietrich Weber and Anoop Gupta
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

November 16, 1988

Abstract

A fundamental problem that any scalable multiprocessor must address is the ability to tolerate high latency memory operations. This paper explores the extent to which multiple hardware contexts per processor can help to mitigate the negative effects of high latency. In particular, we evaluate the performance of a directory-based cache coherent multiprocessor using memory reference traces obtained from three parallel applications. We explore the case where there are a small fixed number (2-4) of hardware contexts per processor and the context switch overhead is low. In contrast to previously proposed approaches, we also use a very simple context-switch criterion, namely a cache miss or a write-hit to shared data. Our results show that the effectiveness of multiple contexts depends on the nature of the applications, the context switch overhead, and the inherent latency of the machine architecture. Given reasonably low overhead hardware context switches, we show that two or four contexts can achieve substantial performance gains over a single context. For one application, the processor utilization increased by about 65% with two contexts and by about 100% with four contexts.

1 Introduction

As shared-memory multiprocessors are scaled (the number of processors is increased), there will invariably be an increase in the latency of memory operations. While local memory references need not have higher latency, remote memory operations will encounter higher latency because of the larger physical size of the machine, if not for any other reason. Consequently, there will always be times when a processor sits idle, waiting for some remote operation to complete [2,11]. If more than one context resides on each processor, and context switch overhead is low, this idle time can be used by additional contexts. Typically each context corresponds to a process from one parallel program.

In this paper, we evaluate the utility of multiple contexts per processor for a directory-based cache coherent multiprocessor [1]. While the idea of using multiple hardware contexts per processor is itself not new, we believe our scheme is simpler to implement than other proposals [4,8,11,19,21] (discussed in Section 5). In our scheme, each processor contains a small fixed number (2-4) of hardware contexts with independent register sets to enable short context switch times. We also use a very simple context switch criterion, which is to switch contexts on a cache miss or on a write-hit to read-shared data or when a watchdog counter of 1000 expires.[1] This simple scheme helps keep context switch overhead low, because the decision to switch or not can be made in a single cycle.

Our multiple context scheme is evaluated using multiprocessor memory-reference traces obtained from three applications [13,16,20]. The results indicate that multiple contexts can achieve substantial gains in processor utilization. In some cases processor utilization is increased by 65% with two contexts and by 100% with four contexts.

The rest of the paper is organized as follows. The next section presents the architecture and simulator used in this study. We also introduce the applications and the method employed to gather the reference traces. Section 3 gives general results for the three applications. After that we present a number of issues concerning multiple contexts. This section also gives the results of the simulations. Finally, we have the related work, discussion and conclusion sections.

2 Architectural Assumptions and Simulation Environment

In this section, we discuss the architectural assumptions that we make and describe the simulation environment that we used to obtain our results. We also describe the applications used in this study and the performance metric employed to evaluate the multiple context scheme.

2.1 Base Architecture and Simulator

Figure 1 shows the basic architecture that we assume in this paper. The architecture consists of several nodes linked together by an interconnection network. Each node has a processor, a physical cache, and its share of the global memory. It is connected to the network through the directory (DIR) and network interface (N.I.). The processors may have one or more contexts. The caches are kept consistent using a directory-based cache coherence protocol as discussed in [1]. We study the performance as a function of several parameters such as the number of contexts, the context switch overhead, the latency of the network, and so on. Performance results as a function of the above parameters are given in Section 4.

[1] The watchdog counter is introduced to prevent one context from hogging a particular processor. This ensures that no context runs for longer than 1000 cycles at a time, preventing starvation and deadlocks.

Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results

Wolf-Dietrich Weber and Anoop Gupta
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

[redacted]

November 16, 1988

Abstract

A fundamental problem that any scalable multiprocessor must address is the ability to tolerate high latency memory operations. This paper explores the extent to which multiple hardware contexts per processor can help to mitigate the negative effects of high latency. In particular, we evaluate the performance of a directory-based cache coherent multiprocessor using memory reference traces obtained from three parallel applications. We explore the case where there are a small fixed number (2-4) of hardware contexts per processor and the context switch overhead is low. In contrast to previously proposed approaches, we also use a very simple context-switch criterion, namely a cache miss or a write-hit to shared data. Our results show that the effectiveness of multiple contexts depends on the nature of the applications, the context switch overhead, and the inherent latency of the machine architecture. Given reasonably low overhead hardware context switches, we show that two or four contexts can achieve substantial performance gains over a single context. For one application, the processor utilization increased by about 65% with two contexts and by about 100% with four contexts.

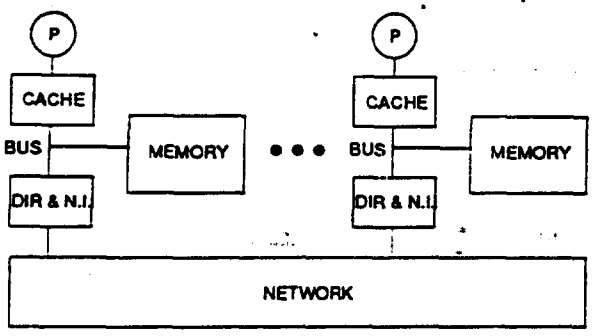

Figure 1: Architectural model

We use a trace-driven simulator, written by Truman Joe at Stanford, that emulates the above architecture to evaluate the effectiveness of multiple contexts. In the single context per processor case, the simulator works as follows. Before starting the simulation, we first divide the interleaved reference stream generated by the tracing program into separate streams for individual processors. Then, one reference stream is assigned to each of the processors. At every simulated clock cycle, each active processor reads the next reference from its associated reference stream. If the reference hits in the cache[2] the processor remains active and will issue another reference from the stream on the next clock tick. However, if it misses or a write to read-shared data occurs, it context switches. The cache sends a request over the network to fetch the missing line and/or update the state of the other caches in the system. During the period of time that the cache request is waiting to be satisfied, the processor remains in a suspended state and does not generate any more references.

In case of multiple contexts per processor, we have multiple memory reference streams associated with each processor — one for each context. At any given time only one of these contexts is active and the memory references come from that stream. However, when the active context enters the suspended state due to a cache miss or a write hit on read-shared data, a context switch occurs. The processor stays idle for the time required to perform the context switch. After that, memory references are issued from the newly activated context. If more than one context is ready when the active context blocks, a round-robin scheduling scheme decides which context is to be activated next.

The simulator that we use is quite detailed in that it models contention for the memory modules, for the bus on which the memory modules reside, for the directory associated with each node, and for the interconnection network. It is also possible to vary the delays associated with each of the above modules. We note that the interconnection network assumed in our simulations is a crossbar switch, but it could be any point-to-point network (e.g., grid [18], butterfly [3], omega [15]) depending on the number of processors we wished to interconnect. For the default parameters that we used (shown in Table 1), a remote read takes 27 cycles and a remote write takes 19 cycles with no contention. The local operations take 16 and 13 cycles respectively. With contention these numbers can grow to as large as 100 cycles in our simulations.

The simulator is driven by multiprocessor memory reference traces. Since the traces include 16 reference streams, we are limited to four processors if we wish to explore four contexts per processor. For runs with fewer than four contexts, only some of the reference streams were used. We model the scaling of the machine architecture to a larger number of processors by increasing the latency in the underlying network (see Section 4.3). We also vary the context switch overhead and the number of contexts per processor. Section 4 will present the issues involved and the results obtained.

[2]For writes, the location has to be owned in addition to being present in the cache.

Operation	Time
Memory Latency	6 cycles
Bus Transfer	4 cycles
Switch Latency	2 cycles
Switch Transfer	4 cycles
Directory Lookup	2 cycles

Table 1: Default Parameters for Simulator

One inaccuracy in our simulator is that we assume an infinite cache for each processor.[3] Thus, we do not model the interference in the caches when there are multiple contexts per processor. It is not clear, though, whether the sharing of caches is an advantage or a disadvantage. If the caches are small, interference might be a serious problem. With fairly large caches, however, the pre-fetch achieved by contexts working on the same shared data could actually be beneficial.[4] The caches in the architecture presented here are expected to be large as they serve as the main source of remote code and data.

2.2 Traces and Applications

The multiprocessor traces used in our simulations were gathered on a VAX 8350, using a combined hardware/software scheme [5]. Basically, the tracing works as follows. We spawn as many processes as the application desires under the control of a master process. The master process then single steps the application processes in a round-robin manner. After each step, it records all references made by the application processes. For each reference, the number of the processor producing it, the address of the reference and its type (read/write/ifetch) are recorded. The traces that we use correspond to 16-processor runs.

The traces used were obtained from three applications: LocusRoute, MP3D and P-Thor. LocusRoute [16,17] is a standard cell global router. While the tasks spawned by it are quite coarse in granularity (each may execute around 100,000 instructions), its central data structure (a global cost array) is shared at a fine granularity. MP3D [13] is a 3-dimensional particle simulator that determines the shock waves generated by a body flying at high speed in the upper atmosphere. It uses distributed loops for parallelization (each loop executes around 250 instructions) and it is a typical example of parallel scientific code. P-Thor [20] is a parallel logic simulator that uses the Chandy-Misra distributed simulation algorithm. Each parallel subtask (a component evaluation) in P-Thor takes about 300 instructions to execute.

[3]We are working on an a new version of the simulator that will remove this restriction.

[4]Note that in our execution model, several processes from the same application are using the multiple contexts. Thus the amount of shared data can be significant.

2.3 Performance Measure

The main figure of merit used in evaluating multiple contexts in this paper is *processor efficiency*. This is defined as the number of cycles spent doing useful work over the total number of cycles. Of course, the maximum is one reference per processor per cycle for 100% efficiency. The more time the processors spend idle, waiting for remote reads and writes, the lower the overall processor efficiency. In our simulations, we ran the system for a total of 500,000 clock cycles, and then counted the number of memory references consumed from the traces to get the efficiency.

3 General Results

In this section we present some general results obtained with the simulator. These results give an overall idea of the differences in behavior of the three applications. They also show the effect of increasing the switch latency on the read and write latencies seen by the processors. The numbers are for a 4-processor system with one context per processor. The tables below give data about the run lengths and latencies for the three applications. Run length is defined as the number of simulator cycles between each cache miss.[5] Read and write latencies are the number of cycles required to satisfy the cache miss.

Results for switch latencies of 2 and 16 cycles are presented. A switch latency of only two cycles is close to the minimum that can be achieved with any type of network. The switch latency of 16 represents the latencies that might be expected in a larger multiprocessor with many more nodes.

	Run Length		Read Ltncy		Write Ltncy	
Application	Avg	Med	Avg	Med	Avg	Med
MP3D	20	14	25	27	19	19
P-Thor	91	18	24	27	17	19
LocusRoute	107	45	24	27	17	19

Table 2: General application results with switch latency of 2 cycles

	Run Length		Read Ltncy		Write Ltncy	
Application	Avg	Med	Avg	Med	Avg	Med
MP3D	18	14	51	55	33	33
P-Thor	58	17	48	55	31	33
LocusRoute	100	44	54	55	37	30

Table 3: General application results with switch latency of 16 cycles

Both average and median values are given to convey more information concerning the distribution of the run-lengths and latencies. Median values are more representative in characterizing the typical run-length. In LocusRoute, for example due to a few very long runs the averages are high even though the median values are much lower.

MP3D has the shortest run-length and longest latencies. There is a lot of global data traffic in MP3D and this leads to frequent misses, i.e. short run lengths. LocusRoute, on the other hand has very long run-lengths. The large size of the tasks and their relative independence allows for large portions of code that execute out of the cache without any misses. The latencies are close to the minimum expected for this architecture. P-Thor is somewhere in between the other two applications.

As the switch latency increases, the read and write latencies grow as well. Reads are affected more because they require a two-way transaction and so the higher latency is incurred twice. Run lengths should be unaffected by the increased latency, but in fact we do see a slight decrease in run lengths as the switch latency increases. This is probably due to a cold-start effect of the caches. Run-lengths near the beginning of the reference streams are shorter on average, because more cache misses are incurred.

4 Issues and Results

We wish to explore several questions concerning the performance of multiple contexts:

- How many contexts are required to achieve good processor utilization?
- How does the context switch overhead affect the performance?
- What is the effect of increasing the switch latency?
- When to switch contexts?
- How much does the performance vary with application?

This section explores all of these issues and presents results. We show graphs of processor efficiency. In each graph, we are plotting the number of active cycles over the total number of cycles against the switch latency of the architecture. We show efficiencies for one, two and four contexts. Different context switch overheads are presented on different graphs. Figures 2–4 show results for MP3D. Figures 5–7 give results for P-Thor and Figures 8–10 show results for LocusRoute.

4.1 Number of Contexts

Depending on the single context processor efficiency, it may or may not be worthwhile to use two, four or more contexts. Note that the single-processor efficiency is basically a function of the cache miss rate and the read and write latency for the architecture. For LocusRoute (Figures 8–10) the processor efficiency is already very high (about 90%) with a single context and little performance can be gained by adding more contexts. As a matter of fact, if the context switch overhead is high, four contexts do worse than one (Figure 10). MP3D on the other hand (Figure 2), has single context performance near 50% and achieves substantial gains with more contexts (efficiency is 77% with 2, 94% with 4).

As expected, the graphs show diminishing marginal returns as the number of contexts is increased (see Figure 3 for example). In every case going from one to two contexts yields a greater benefit than going from two to four contexts. A small number of contexts is also preferable because it allows simpler

[5] Both here and in the rest of the paper, by *cache miss* we actually mean references that can not be satisfied by the cache alone and need to access the memory, or the network, or both. These include regular cache misses but also write-hits to read-shared data. In the latter case, the network needs to be accessed to invalidate that location from other caches and to gain ownership of that cache line.

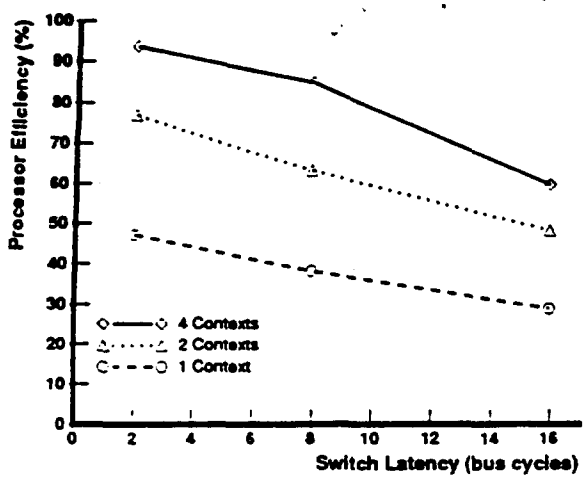

Figure 2: MP3D: Context Switch Overhead 1 Cycle

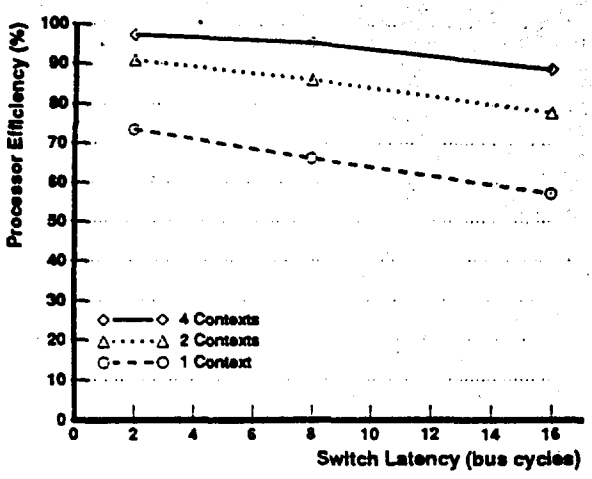

Figure 5: P-Thor: Context Switch Overhead 1 Cycle

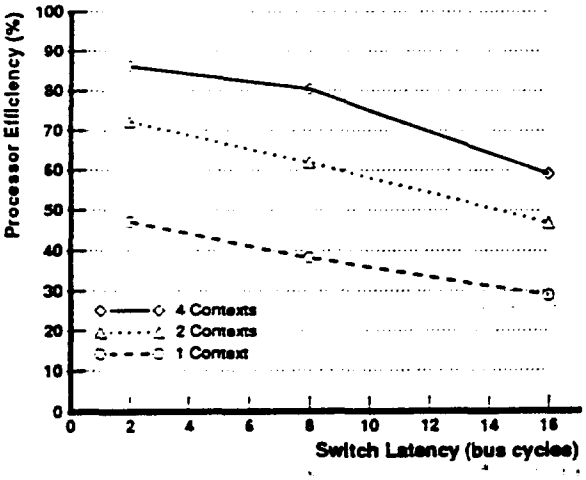

Figure 3: MP3D: Context Switch Overhead 4 Cycles

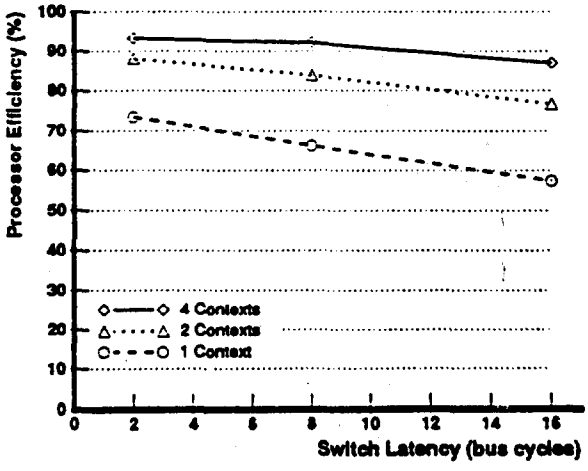

Figure 6: P-Thor: Context Switch Overhead 4 Cycles

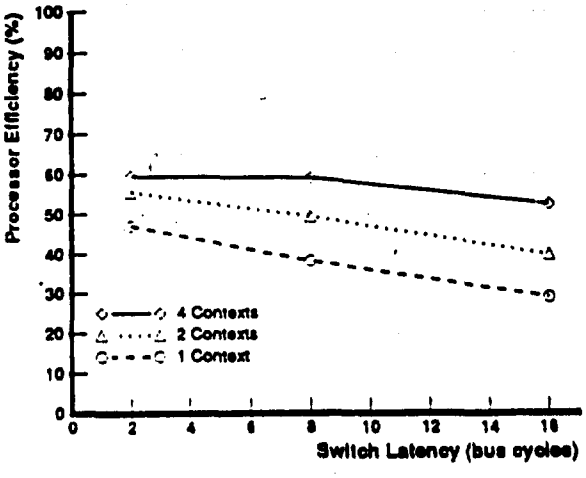

Figure 4: MP3D: Context Switch Overhead 16 Cycles

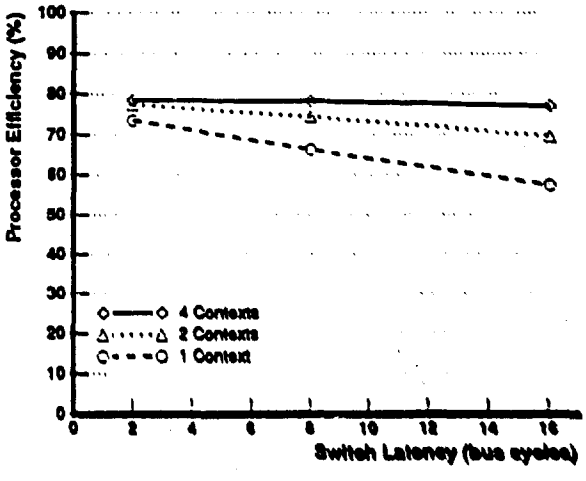

Figure 7: P-Thor: Context Switch Overhead 16 Cycles

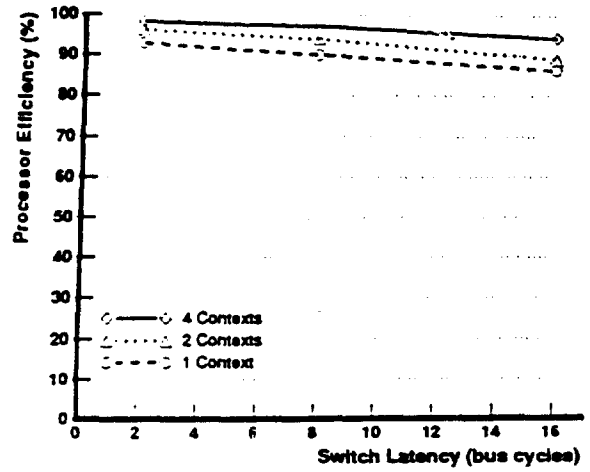

Figure 8: LocusRoute: Context Switch Overhead 1 Cycle

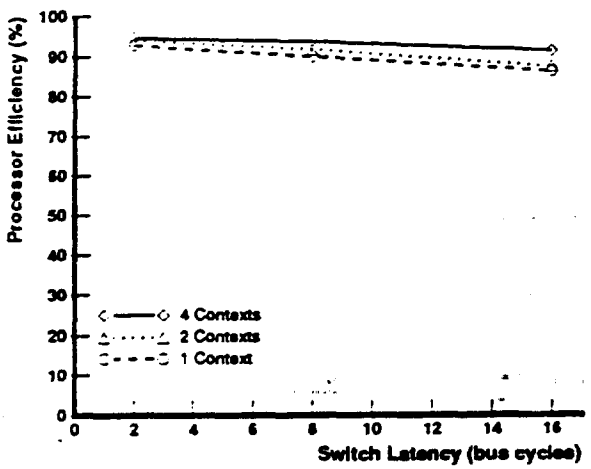

Figure 9: LocusRoute: Context Switch Overhead 4 Cycles

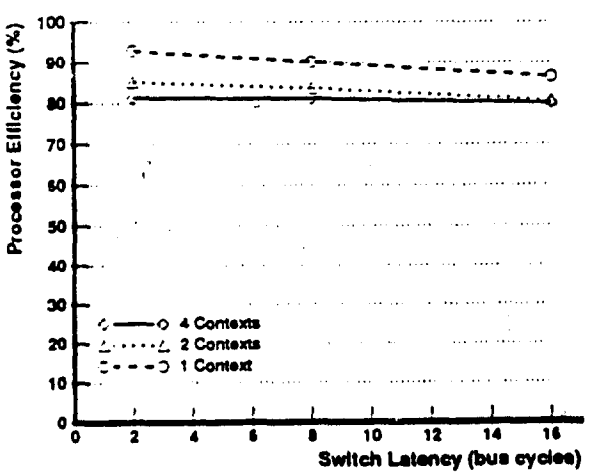

Figure 10: LocusRoute: Context Switch Overhead 16 Cycles

hardware. With a larger number of contexts, a penalty in the cycle time of the processor or an increase in context switch overhead may be inevitable. Also, a large number of contexts requires a large number of processes. Many applications may not be able to support such a large number of processes.

4.2 Context Switch Overhead

The context switch overhead depends on the number of contexts kept in hardware, the amount of state kept for each context, and the amount of hardware dedicated to context switching. We explore context switch overheads of 1, 4 and 16 cycles. A single cycle overhead can be achieved by keeping multiple copies of the pipeline registers and being able to swap in the whole state in a single cycle.[6] If the pipeline has to be drained and filled, a 4-cycle overhead is reasonable. Both of these options require multiple register banks, one for each context. If we want to load and store the registers to some fast local memory, we have to allow at least 16 cycles. It is clear that the hardware is more complex if we require the context switch to be faster. Of course, beyond some overhead value, multiple contexts do not help any more, since a long latency operation will complete before the context switch is achieved.

As expected, the results show that the effect of increasing the context switch overhead reduces the benefit achieved by having multiple contexts. Note that the single context graph line is identical for various context switch overheads (see Figures 2–4 for example), since there is no context switching in that case. When the context switch overhead is 16, none of the programs are gaining much processor efficiency with increased contexts. MP3D achieves a 12% increase in efficiency with 4 contexts (Figure 4), P-Thor gains only 5% (Figure 7) and LocusRoute actually looses 12% (Figure 10). For multiple contexts to be useful, the context switch overhead will have to be kept low, preferably on the order of a few cycles.

4.3 Latency

The amount of latency incurred in remote operations is important for the effectiveness of processors with multiple contexts. With very low latencies, context switch overhead may be too large to allow multiple contexts to achieve any performance gain. As the latency increases, the single context processors do increasingly poorly because more and more processor time is spent idle. This is where multiple contexts can help. As seen in Figures 5–7, the relative value of multiple contexts increases as the latency increases. In other words, a processor with multiple contexts will suffer less efficiency degradation due to high latencies than a single context processor.

One reason for varying switch latency in our evaluation of multiple contexts is to explore different types of architectures. A grid network, for example, is expected to have a much larger latency than a crossbar switch. At the same time the higher latencies can correspond to larger multiprocessors. As more processors are added to a parallel machine, the latencies increase due to deeper networks or more complex switches. Larger latencies present a greater opportunity for multiple contexts, because the single context efficiency is lower. At the same time we note that it is still possible to achieve very high efficiencies with just a few contexts. For example, with

[6] Alternatively multiplexors could be used to switch between multiple pipeline state copies.

a switch latency of 16 cycles, latencies are on the order of 50 and 30 cycles for reads and writes respectively (see Section 3). A network large enough to have this high a latency could well support several hundred processors. Yet processor efficiencies stay high for this latency (60% for MP3D, 89% for P-Thor and 94% for LocusRoute). The point is that even as multiprocessors grow and latencies increase, processors with just a few contexts achieve very good utilization.

4.4 When to Switch Contexts

Ideally, one would like to switch contexts whenever the context switch overhead is less than the latency of the operation being performed. Of course external operations may take longer or shorter depending on the congestion in the machine, and there is no easy way to predict how long a given operation will take. We thus choose the easiest context switch criterion: switch on any operation that requires a main memory access, either in the same cluster or remotely. Switching only on *remote* operations requires extra hardware, but is a feasible alternative if context switch overhead is relatively high. If a context switch takes 16 cycles, and local operations also take on the order of 16 cycles to complete, it does not make sense to initiate a context switch on every local operation.

Two of the applications had frequent memory accesses, but LocusRoute processes had long streaks of executing out of the cache. In order to prevent one context from hogging a particular processor we introduce a watchdog counter that pre-empts the current context after 1000 cycles. This ensures that no context runs for longer than 1000 cycles at a time, thus allowing all contexts on a particular processor to make progress.

4.5 Applications

The three applications exhibited very different behavior. LocusRoute and P-Thor have relatively little global traffic, whereas MP3D has a lot. While 1.8% of LocusRoute instructions cause references to shared data, this number is close to 12% for MP3D. This explains why the run-lengths presented in Section 3 are so different for the three applications. At the same time LocusRoute has very good caching behavior and very little interference between processes. Thus LocusRoute achieves very high efficiencies (around 90%), even with single context processors (see Figures 8–10). Very little can be gained by adding extra contexts.

P-Thor achieves 50-70% utilization with single contexts (see Figures 5–7). This can be boosted effectively by adding more contexts. Not only is efficiency increased as more contexts are added, but the processors also become more immune to the effect of high latency operations. This is seen by the spreading of the curves as the latency increases.

MP3D has a large amount of global traffic. When the switch latency increases, the switch becomes the bottleneck and it limits the gains achieved by multiple contexts. While some performance gain is achieved, the relative benefit of multiple contexts is greater for lower latencies. Note how the different context lines converge as the switch latency increases in Figures 2 and 3.

5 Related Work

The idea of multiple hardware contexts per processor in itself is not new. In this section we discuss how our approach differs from earlier proposals and present some advantages and disadvantages. We begin with the Alto personal computer from Xerox [21] which provided multiple hardware microcode-level contexts, allowing the CPU to be shared between the instruction set interpreter and the I/O devices. The contexts were statically assigned to devices and were not available to general user processes. The aim of the multiple contexts was to make the power of the processor readily available for time critical I/O processing, a task that is frequently delegated to separate processors in more recent designs. Unlike our motivation, the issue was not to hide memory latency from a very fast processor.

The HEP multiprocessor from Denelcor [19] also provided multiple hardware contexts per processor. Unlike the Alto, the contexts were available to arbitrary user processes. The processes shared a large set of registers and on each cycle an instruction from a different process was executed. A minimum of 8 *active* processes (those processes that are not waiting for a memory reference to complete) were needed to keep the execution pipeline full. The HEP machine tolerated memory latency well, but its main drawback was that a single process could get at most 1/8 of the pipelined processor. In order to keep the pipeline full, a large number of processes were needed. This is in stark contrast to modern pipelined processors [6,14] where a single process almost fully utilizes the pipelined processor. Now the HEP scheme would not be a problem if all applications could be split into an arbitrarily large number of processes. However, this is often not possible in practice as there may not be enough intrinsic parallelism in the application [7], or because doing so greatly increases the amount of overhead.

More recently, Iannucci [11] has proposed using multiple contexts for his hybrid data-flow/von Neumann machine. Each processor consists of a hardware queue of enabled *continuations*. The continuations are very small in size (containing just the program counter and the frame base-register), and the hardware can switch between them in a single cycle. However, to make this single cycle switch possible, processor registers are not saved on a context switch. Consequently, the software is structured so that it does not rely on registers being valid between potential context switch points. The switch points are synchronizing references, where a read to a location tagged *empty* results in that continuation being suspended. In our view, the disadvantages of Iannucci's approach are the following. First, processes can not make full use of the register sets, given that the run-lengths (the number of instructions executed between switch points) are very small [11] and registers are not preserved in between. We believe that extensive use of registers is absolutely critical to the performance of modern processors [6]. Second, a processor that supports a large number of continuations (contexts) in hardware, keeps track of which ones are enabled and uses a complex criterion for deciding which continuation to issue the next instruction from [12], is very complicated. We believe such a processor will have a significantly more complex pipeline and much larger area than a simple RISC processor. Consequently, the cycle time of such a machine would be slower than that of modern RISC processors. Thus the hybrid machine has to make up the large factor that it loses over conventional microprocessors, before it becomes competitive. On the other hand, the scheme that we propose does not lose

anything over modern RISC processors. In fact, it is possible to take multiple commercially available RISC processor chips (e.g., Motorola 88000 processor and cache chips) and connect them so as to simulate multiple contexts.

We now consider the MASA architecture proposed by Bert Halstead [8]. In this architecture each processor has a fixed number of hardware *task frames*. Each task frame is capable of storing a complete process context and consists of a set of auxiliary registers (like the program counter) and a set of general purpose registers. Since the number of processes may exceed the number of task frames, the process contexts are allowed to overflow into memory.[7] On each cycle, a context in the *enabled* or *ready* state may issue an instruction. However, once a process issues an instruction, it can not issue another instruction until the previous instruction has completed. Thus, in its current form, a process on MASA can get only 1/4 (inverse of pipeline depth) of the pipelined processor's performance. As discussed above for HEP, this is a major drawback. Halstead and group recognize it [8] and are exploring ways to remove this restriction.

We now discuss a more subtle but fundamental difference between the Iannucci and Halstead schemes and our scheme. In our scheme, the sole purpose of the multiple hardware contexts is to mitigate the negative effects of memory latency. The number of hardware contexts needed for a particular machine is fixed and depends mainly on the expected cache hit ratio and the memory latency for that architecture. In the Iannucci and Halstead schemes, the context mechanism is instead made to serve two purposes at the same time. It is used to mask memory latency as in our scheme, but it is also used as a hardware task queue. Thus when a parallel subtask is created, it manifests itself as a new context that is then managed and scheduled by the hardware. Since the number of parallel subtasks can be arbitrarily large, mechanisms are needed and provided to handle overflow of contexts. Also, the number of contexts that are needed is large. In our scheme, the issue of subtask management is completely separated and is handled in software. This permits great flexibility, including the possibility to schedule tasks in a manner similar to the Iannucci and Halstead proposals, if a particular application so warrants.[8] Thus instead of using full/empty bits and hardware queuing in I-structure memory [10], we may simulate full/empty bits in software and switch to a different subtask if a piece of data is not ready. It is not obvious which scheme works better. We will be able to tell only when such machines actually get built.

6 Discussion

This section contains the discussion of several topics that relate to the evaluation of multiple contexts as presented in this paper.

One question that we must ask is, what are the real advantages of having multiple contexts? Since processors are cheap, why not simply have a larger number of processors in the multiprocessor? The fallacy in this argument is that, while CPU chips (e.g., MC68030 chips) are relatively cheap, a fast processor is not — a fast processor nowadays has a large amount of cache built out of expensive and fast SRAMs; in addition, there are expensive functional units such as floating point ALUs. Furthermore, each new processor needs an extra port to the network, or to the bus that it is placed on. The extra port increases the depth of the network, or the loading on the bus, thus increasing the latency. Several contexts per processor can share these expensive resources, thus making more efficient use of them.

Another question that arises is how the multiple contexts should be implemented. The multiple contexts do not necessarily have to be implemented on a single chip. In the case where the size of each processing node is small, on the order of a few chips [9], we need to have several contexts on a single chip using duplicated register sets. However, having to design a special processor for a given architecture makes that architecture less practical. So for larger processing nodes, for example where each processor occupies a whole board, it may be quite feasible to use separate processor chips for the different contexts. While simplifying the hardware design effort, this approach duplicates not just the register set but all of the data path and control as well.[9]

There are some software issues to be resolved. In particular, how do you choose which processes to put on a single processor? Since the progress of contexts on any one processor is mutually exclusive, the correct placement of processes on processors may be important. If a given program section requires several contexts to be active in order to make progress, it is best to place these on separate processors.

7 Conclusions

In scalable multiprocessor architectures, processors with a small fixed number of contexts can achieve substantially greater efficiencies than single context processors. In some cases efficiencies increased 65% with two contexts and 100% with four contexts. Best improvements are found in architectures with high latency operations and low context switch overheads. Such high latency operations are to be expected in large-scale multiprocessors. Low context switch overheads can be achieved by having a small fixed number of contexts in hardware and by using a simple switch criterion: the cache miss.

One important difference between our context switch scheme and those proposed in [8,11,19] is that in our scheme the context switch mechanism is separated from the subtask management mechanism. This makes for simpler and faster hardware and allows greater flexibility and application-dependent performance tuning.

We are currently working on more detailed simulations, including the effects of finite caches and cache contention when a miss is satisfied from memory. We are also looking further into the issues and details of implementing our multiple context scheme.

8 Acknowledgements

We would like to thank Truman Joe for allowing us to use his simulator and for helping us understand it and modify it. We also wish to thank Richard Sites at Digital Equipment Corporation, Hudson MA, for supporting Wolf-Dietrich Weber.

[7] Such overflow and underflow operations are quite expensive, and care must be taken to minimize them.

[8] We would normally expect there to be some sort of a distributed task queue to handle the scheduling of subtasks.

[9] An alternative to this scheme which uses multiple CPU chips for contexts is to let all these chips be active all the time, sharing and stalling on the cache as they proceed. We have, as yet, not done any performance evaluation for such a scheme.

Anoop Gupta is supported by DARPA contract N00014-87-K-0828 and by a faculty award from Digital Equipment Corporation. Thanks also to Helen Davis, Margaret Martonosi, Jonathan Rose, Rich Simoni, Larry Soule and Mike Smith for reviewing early versions of this paper.

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *15th International Symposium on Computer Architecture*, 1988.

[2] Arvind and R. A. Iannucci. A Critique of Multiprocessing von Neumann Style. In *15th International Symposium on Computer Architecture*, pages 426–436, 1983.

[3] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance Measurements on a 128-node Butterfly Parallel Processor. In *Intl. Conf. on Parallel Processing*, pages 531–540, 1985.

[4] William J. Dally et al. Architecture of a Message-Driven Processor. *The 14th Annual International Symposium on Computer Architecture*, 189–196, June 1987.

[5] Stephen R. Goldschmidt. Simulating Multiprocessor Memory Traces. December 1987. EE390 Report, Stanford University.

[6] T. Gross, J. Hennessy, S. Przybylski, and C. Rowen. Measurement and Evaluation of the MIPS Architecture and Processor. *ACM TOCS*, 6, August 1988.

[7] Anoop Gupta et al. Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis. *International Journal of Parallel Programming*, 17, 1988.

[8] R. H. Halstead and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *15th International Symposium on Computer Architecture*, pages 443–451, 1988.

[9] J. P. Hayes et al. A Microprocessor Based Hybrid Supercomputer. *IEEE Micro*, 6, October 1986.

[10] S. K. Heller. *An I-Structure Memory Controller (ISMC)*. Technical Report, Massachusetts Institute of Technology, June 1983.

[11] R. A. Iannucci. Toward a Dataflow / von Neumann Hybrid Architecture. In *15th International Symposium on Computer Architecture*, pages 131–140, 1988.

[12] Robert A. Iannucci. *A Dataflow / von Neumann Hybrid Architecture*. PhD thesis, Massachusetts Institute of Technology, 1988.

[13] Jeffrey D. McDonald and Donald Baganoff. Vectorization of a Particle Simulation Method for Hypersonic Rarified Flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[14] D. Patterson. Reduced Instruction Set Computers. *Comm. ACM*, 28, January 1985.

[15] G.F. Pfister, W.C. Brantley, et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *International Conference on Parallel Processing*, IEEE, 1985.

[16] Jonathan Rose. LocusRoute: A Parallel Global Router for Standard Cells. In *Design Automation Conference*, pages 189–195, June 1988.

[17] Jonathan Rose. The Parallel Decomposition and Implementation of an Integrated Circuit Global Router. In *Proc ACM SIGPLAN PPEALS*, pages 138–145, July 1988.

[18] Charles L. Seitz, William C. Athas, Charles M. Flaig, Alain J. Martin, Jakov Seizovic, Craig S. Steele, and Wen-King Su. The Architecture and Programming of the Ametek Series 2010 Multicomputer. In *Hypercube Concurrent Computers and Applications*, 1988.

[19] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE*, pages 241–248, 1981.

[20] Larry Soule and Tom Blank. Parallel Logic Simulation on General Purpose Machines. In *Design Automation Conference*, pages 166–171, June 1988.

[21] C. P. Thacker, E. M. McCreight, et al. Alto: A Personal Computer. In C. Gordon Bell Daniel P. Siewiorek and Allen Newell, editors, *Computer Structures: Principles and Examples*, pages 549–572, McGraw-Hill, 1982.